Exploring Functional Slicing in the Design of Distributed SDN Controllers

Yiyang Chang[1], Ashkan Rezaei[2], Balajee Vamanan[2], Jahangir Hasan[3], Sanjay Rao[1], and T. N. Vijaykumar[1]

[1] Purdue University
[2] University of Illinois at Chicago
[3] Google Inc.

Abstract. The conventional approach to scaling Software-Defined Networking (SDN) controllers today is to partition switches based on network topology, with each partition being controlled by a single physical controller, running all SDN applications. However, topological partitioning is limited by the fact that (i) performance of latency-sensitive (e.g., monitoring) SDN applications associated with a given partition may be impacted by co-located compute-intensive (e.g., route computation) applications; (ii) simultaneously achieving low convergence time and response times might be challenging; and (iii) communication between instances of an application across partitions may increase latencies. To tackle these issues, in this paper, we explore *functional slicing*, a complementary approach to scaling, where multiple SDN applications belonging to the same topological partition may be placed in physically distinct servers. We present *Hydra*, a framework for distributed SDN controllers based on functional slicing. Hydra chooses partitions based on convergence time as the primary metric, but places application instances across partitions in a manner that keeps response times low while considering communication between applications of a partition, and instances of an application across partitions. Evaluations using the Floodlight controller show the importance and effectiveness of Hydra in simultaneously keeping convergence times on failures small, while sustaining higher throughput per partition and ensuring responsiveness to latency sensitive applications.

1 Introduction

Software-Defined Networking (SDN) is becoming prevalent in datacenter and enterprise networks [12, 11]. The central idea behind SDN is to consolidate control plane functionality (e.g., routing, access control) at a *logically* centralized controller which monitors and manipulates network state [8, 19]. An SDN controller for a small network with hundreds of switches could be hosted on a single physical server. However, as networks grow in size and functionality, the controller's compute and memory requirements exceed one single server's capacity. Therefore, large datacenter and enterprise networks distribute the controller functionality over multiple servers or VMs [7, 9, 21, 16].

Real SDN deployments typically consist of several tens of SDN applications for diverse network tasks such as routing, load-balancing, security, and Quality of Service (see Figure 1). Because these applications handle different events (e.g., link failure vs. path lookup) and perform diverse functions, they impose varying demands on the underlying machine resources. We broadly classify them into three groups:
(1) *Real-time* applications that periodically refresh network state (e.g., link manager, heart-beat handler) expect a response within a timeout interval; failing to respond before deadline would trigger expensive false alarms (e.g., a spurious link failure).
(2) *Latency-sensitive* applications that are invoked during flow setup (e.g., path lookup, bandwidth reservation or QoS) are in the critical path of applications and directly impact flow completion times. Therefore, it is crucial to reduce their latency. However, they don't have a hard deadline constraint.
(3) *Computationally intensive* applications such as shortest-path computation are triggered less often due to infrequent events such as link failures. But when triggered, these applications exert substantial pressure (load spikes) on compute and memory. Convergence time, which is the time required for *global* state convergence (e.g., time required for find alternate paths in all partitions after a link failure in one partition), is an important metric for these applications.

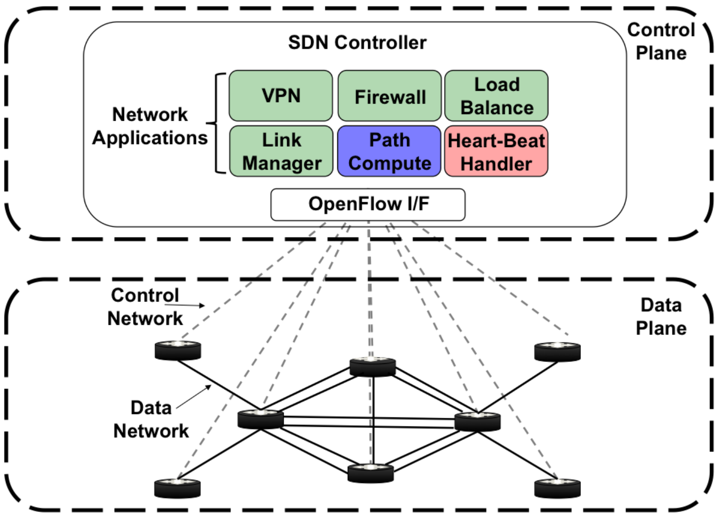


Fig. 1: An example SDN network

Designing a distributed control plane that scales well with network size and application heterogeneity is an important problem. The conventional approach to scaling SDN deployments [7, 9, 21, 16] is *topological slicing* where the network topology is partitioned across multiple controller instances. Each controller instance, which runs on a single server machine, co-locates all applications and handles all events from a network partition containing a subset of switches.

Topological slicing suffers from a few shortcomings:
(1) Because topological slicing co-locates all applications, finding the best partition size that satisfies all applications (i.e., missed deadlines for real-time applications, latency for latency-sensitive applications, and convergence time for computationally intensive applications) is hard. For instance, co-locating computationally intensive applications with other applications may require smaller partition sizes (i.e., higher number of partitions) in order to satisfy resource constraints on the server machine. However, increasing the number of partitions would likely worsen convergence time for route recomputation on failures. Also, latency-sensitive applications such as bandwidth reservation and QoS may require communication across multiple instances of the application running across partitions at flow setup time, potentially leading to an increase in packet-in response times as the number of partitions increase. Finally, there could be other administrative constraints on partition sizing (e.g., a unit within an organization may want to have a separate controller instance). In summary, while there is substantial diversity among applications, topological slicing is agnostic of the different applications' requirements, and, therefore, does not scale well. (2) Topological sizing hurts real-time and latency-sensitive applications. Because computationally intensive applications are susceptible to load spikes, co-locating computationally intensive applications with real-time and latency-sensitive applications adversely affects their latencies (real-time applications are most affected by co-location) as we show in Figure 7.

We propose *functional slicing*, an approach that complements topological slicing by splitting different control-plane functions across multiple servers. Functional slicing adds a new dimension to the partitioning problem and provides more freedom for placement of applications on different servers. With functional slicing, a switch may forward different events to different controllers (e.g., one could install a forwarding rule at the switch for each event or have the original server forward the events to other servers). With functional slicing, we can optimize the number of partitions to minimize *only* convergence time, without violating administrative constraints and without affecting real-time or latency-sensitive applications.

While functional slicing offers *one more degree of freedom* for partitioning the control plane, it complicates placement. For instance, placing control-plane functions that are in the critical path in different machines would lead to longer flow completion times (i.e., the overhead of crossing machine boundaries would increase response times for packet-in messages during path setup). Therefore, our placement algorithm must be aware of the dependencies between the different control-plane functions.

We present *Hydra*, a framework for partitioning and placement of SDN control-plane functions in different servers. *Hydra* leverages *functional slicing* to increase flexibility in partitioning and placement. Moreover, Hydra's partitioning is *communication-aware* – Hydra considers the communication graph to avoid placing control-plane functions that are in the critical path in different machines. We first formulate the placement of application instances across partitions as an optimization problem with the objective of minimizing the latency of latency-sensitive applications that are in the critical path, subject to resource constraints (i.e., number of servers, CPU and memory per server). We then reduce our formulation to a multi-constraint graph partitioning problem and solve it using well-known heuristics [15]. To shield real-time applications from other applications, Hydra uses thread prioritization. Hydra assigns the highest priority to threads of real-time applications and next highest priority to latency-sensitive applications, while separating computationally intensive applications from the other two categories. Most applications fall in only one of these three categories (e.g., shortest-path calculation is computationally intensive but is neither latency sensitive nor real time). However, if an application belongs to more than one category, we could consider the application in its most critical category and still use Hydra for partitioning and placement.

Hydra is relevant for both reactive controllers (where rules are installed after examining the first packet of each flow), and proactive controllers (where rules are pre-installed in switches) [5]. Our optimization formulation is agnostic to the choice of the model. Our formulation considers the packet-in rates, which may be high for reactive SDNs and low for pro-active SDNs, and the rates, among other factors, influence the best partition chosen by *Hydra*. Our evaluation shows a range of packet-in rates to capture a continuum of this choice.

In summary, we make the following contributions:

• We propose *functional slicing*, which adds a new dimension to partitioning and provides more flexibility.

• We introduce a *communication-aware* placement algorithm that leverages functional slicing and avoids its potential shortcomings.

• We evaluate *Hydra* using Floodlight [2] controller and show the effectiveness of *Hydra's* key techniques – functional slicing, communication-aware placement, and prioritization.

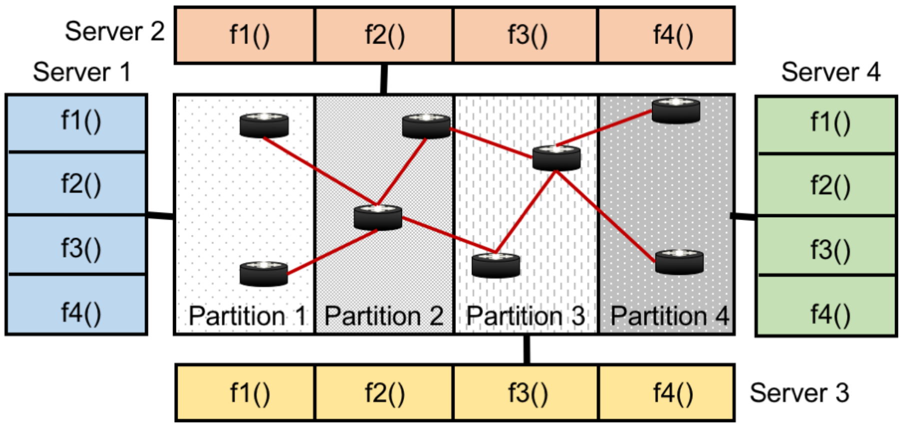
The rest of the paper is organized as follows. Section 2 presents an overview of Hydra's approach, and Section 3 delves into the details of Hydra. Section 4 describes our experimental methodology and Section 5 presents our results. Section 6 discusses related work. Finally, Section 7 concludes the paper. This paper is an extended version of [4]. This extended version includes a section on modeling convergence time (section 3.2) and presents more results on sensitivity.

2 Background and Hydra's Rationale

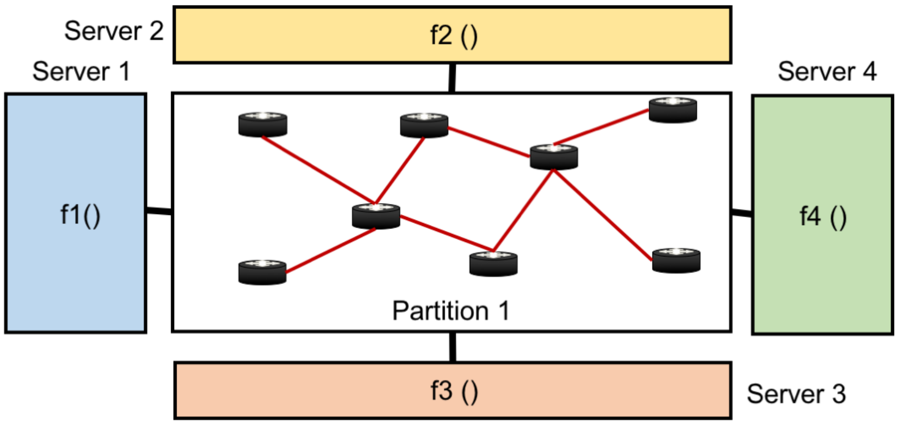
We begin by discussing alternative ways to scale SDN controllers, and present Hydra's approach and rationale:

2.1 Topological partitioning

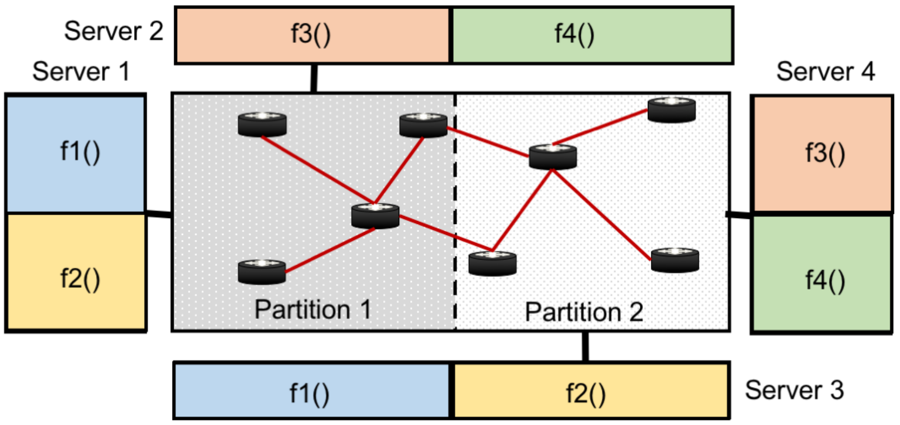
Current distributed controllers [7, 9, 21, 16], de facto assume topological partitioning of the network into multiple controller domains, with one controller in-



(a) Topological slicing



(b) Pure functional slicing



(c) Hybrid slicing with Hydra

Fig. 2: Approaches to partitioning controller functionality.

stance per domain. Each controller instance runs all the control-plane applications (e.g., topology modules, heart-beat handler that monitors switch failures) but handles events only from the switches in its own partition. Figure 2a shows an example of topological partitioning where each partition contains two switches and the four applications (*f1* through *f4*) run on each controller. While topological partitioning helps with scaling, the sustainable throughput is still limited by the fact that the compute and memory capabilities must be sufficient to handle all applications in that partition. Increasing the number of partitions to reduce partition sizes may not be feasible due to network administrator constraints and since this may potentially increase route convergence time when recomputing paths on a switch or link failure. Finally, state changes in any partition of an application may need to be propagated to other partitions in order to maintain consistency of the application's global network state, and flow set up (e.g., for a QoS application) may involve communications across application instances located in different partitions.

2.2 (Pure) Functional slicing

Functional slicing partitions the control-plane functions belonging to the same topological partition and places the functions in different servers. Figure 2b shows an example of functional slicing for the same network as in Figure 2a. The example shows the four functions *f1()* through *f4()* split across four controllers each of which covers the entire network (i.e., all the four topological partitions in Figure 2a). While this tackles some of the issues with topological partitioning, the sustainable throughput may now be bottlenecked by the most demanding application. Further, *pure* functional slicing may worsen the latency to handle critical packet-in events because the control-plane functions needed to handle each such event may be spread across multiple machines (i.e., kernel overheads and networks delays would lie in the critical path of packet-in event-handlers).

2.3 Hydra's approach: A hybrid of topological and functional slicing

With Hydra, we explore a hybrid scheme that employs a combination of topological and functional slicing to reduce both convergence times and packet-in processing latencies. Figure 2c shows an example of our hybrid slicing for the same network as in Figure 2a. The example shows two topological partitions. Each controller and two functional partitions of each of the topological partitions, so that only two servers for each function have to converge as opposed to the four servers in topological partition in Figure 2a. At the same time, an event involving all four functions needs communication only between two servers as opposed to four servers in functional slicing in Figure 2b.

While Hydra separates computationally-intensive applications (i.e., path recomputation) from the other two categories, Hydra shields real-time applications (e.g., heart-beats) from latency-sensitive applications (e.g., path lookup) using *thread prioritization*. Hydra assigns the highest priority to real-time applications and second highest priority to latency-sensitive applications.

3 Hydra

In this section, we discuss Hydra's *communication-aware* placement algorithm. Recall that Hydra leverages *functional slicing* to calculate the number of partitions that minimizes convergence time, without negatively impacting real-time and latency-sensitive applications or violating administrative constraints.

3.1 Finding the right partition size

In the first step, we compute the number of partitions by considering *only* the most critical computationally intensive application that directly impacts convergence time on failures. Often, the topology (route computation) application is the most critical application. While the exact number of partitions that minimizes convergence time is implementation dependent, in general, as we increase the number of partitions (starting from 1), the convergence time would decrease as the computation gets parallelized across partitions. But, after some point, the communication overheads between parallel computing instances would start to overwhelm the benefits from parallelization. Thus, it is reasonable to expect a U-curve with the best partition size somewhere in the middle. But, Hydra's placement algorithm does not depend on the relationship between convergence time and the number of partitions.

3.2 A simple convergence-time model:

We present a simple model for studying the relationship between convergence time and the number of partitions. For our model, we consider the topology module which re-calculates the routes between all pairs of vertices in the topology graph upon failure. We consider border switches to be those switches that must be traversed when the source and destination lie in different partitions. Intuitively, internal-switch failures have lower cost than border-switch failures. However, the exact convergence cost would depend on the topology and partitioning strategy. Fortunately, most datacenter and enterprise networks have hierarchical topologies (e.g., fat-tree, B-cube) and lead to symmetric partitions with *some* fixed number border switches which greatly simplifies our model. For example, most datacenters have *fat-tree* topologies, with each (or a group) of POD(s) forming a partition. Similarly, most enterprise networks have a *hub-and-spoke* structure, with some switches providing connectivity across many sub-networks (e.g., different parts of the organization), which could have further hierarchy. Therefore, we assume symmetric partitions with a fixed number of border switches per partition in our model.

We start with a network with N switches and P symmetric partitions. Let the total number of border switches be B. Thus, each partition contains $n = \frac{N-B}{P}$ internal switches. The partitions are symmetric and each partition contains k border switches. In general, k could be greater than $\frac{B}{P}$ if some border switches belong to more than one partition. Naturally, each border switch is associated with $\frac{kP}{B}$ partitions. For example, in a network with 20 border switches, 5 partitions, and 5 border switches per partition, one of the 5 border switches in each partition also belongs to one other partition. The total number of switches in each partition is $n + k = \frac{N-B}{P} + k$.

For scalability, large networks are hierarchically organized as inter-connections of many small networks; switches maintain fine-grained routes within its domain but aggregate routes to outside world. When aggregating an entire partition, a border switch could advertise a summary metric such as the average cost to switches in that partition, or the minimum, or maximum. Thus, our model maintains routes to all the internal switches (i.e., fine-grained routes) but aggregates routes to the external world (i.e., other partitions). For example, in Figure 3, switches in partition P1 maintain routes to each of the internal switches within P1 and one for each of the other partitions P2 and P3.

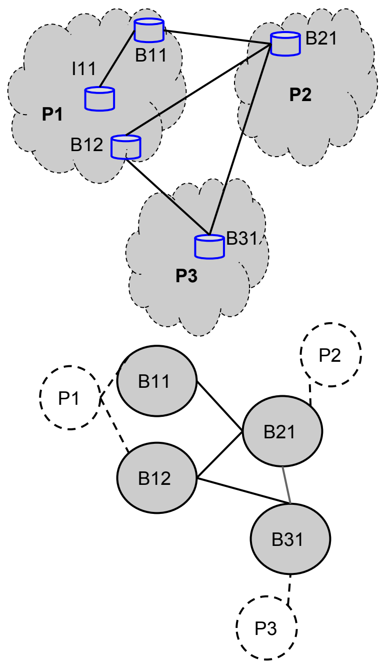


Fig. 3: Example network and its border-switch network

In such a network, route calculation involves three steps:
(1) Internal switches compute internal routes by calculating All-Pair Shortest Paths (APSP) – Dijkstra's algorithm, within each partition. This step has a cost of $(n+k)^2 \log(n+k)$. (Dijkstra's algorithm has a complexity of $O(V^2 logV)$ for V nodes.)
(2) Border switches advertise costs of connecting to their respective partitions (e.g., min-cost path to any of the internal switches within the partition) to

other border-switches, which then run APSP in the border-switch network. The border-switch network has (P+B) nodes – see figure 3. This step has a cost of at most $(P + B)^2 \log(P + B)$. Note that in certain settings, the graph of border routers need not be connected (e.g., our evaluation section discusses such a scenario with fat-tree topologies). In such scenarios, the cost of this step can be discounted.
(3) Internal-switches update their external routes from their border-switches. There are n internal switches, and each internal switch needs to update paths to all partitions, P. For each internal switch and each partition, we pick the lowest cost path from potentially k border switches (and their cost to other partitions). Thus, the cost of this step is nPk.

When a border switch fails, all steps 1–3 are needed in the worst case [6]. However, when an internal switch fails, we could simplify steps 2 and 3. Specifically, in step 2, we *only* need to compute Single Source Shortest-path from the failed partition, which reduces the cost of step 2 from $(P + B)^2 \log(P + B)$ to $(P + B) \log(P + B)$. Similarly, step 3 only needs to compute the routes to the failed partition i.e., cost reduces from nPk to nk. Thus, we have:

$$\begin{aligned} conv\_cost_{border} = {} & (n+k)^2 \log(n+k) \\ & + (P+B)^2 \log(P+B) \\ & + nPk \end{aligned} \tag{1}$$

$$\begin{aligned} conv\_cost_{internal} = {} & (n+k)^2 \log(n+k) \\ & + (P+B) \log(P+B) \\ & + nk \end{aligned} \tag{2}$$

The expected convergence time can be calculated from equations 2 and 1, using the probabilities of internal-switch and border-switch failures. If all switches are equally likely to fail, then the probability of border-switch and internal-switch failure are $\frac{B}{N}$ and $1 - \frac{B}{N}$, respectively.

Thus, the total convergence cost (t_{conv}) is $\frac{B}{N} * conv\_cost_{border} + (1 - \frac{B}{N}) * conv\_cost_{internal}$. Hydra computes the convergence time associated with different partitioning strategies and picks one that is desirable from the perspective of minimizing convergence time. The partitioning strategies are constrained by (i) a specification of the acceptable set of border routers specified by the network administrator; and (ii) the requirement of symmetric partitions. Thus, Hydra's formulation allows us to honor administrative constraints, unlike conventional topological slicing.

3.3 Communication-aware placement: formulation

Hydra takes as input the different (topological) partitions of applications and their demands (CPU and memory), resource constraints (i.e., CPU, memory, and number of servers), and the communication graph to calculate the best placement of the applications' partitions that minimizes latency. We assume that computationally-intensive applications (e.g., path computation) are isolated by placing those applications in separate machines (or VMs); simple prioritization might be sufficient in some cases as well. We cast placement of the applications' partitions as an integer linear programming (ILP) optimization problem. Because our problem is NP hard, we identify a efficient heuristic that can solve it in reasonable time.

Let P be the number of topological partitions, N the number of SDN applications deployed in the network, and S the number of physical servers dedicated for the SDN control-plane. We want to bin-pack $P \times N$ application slices within S server machines such that the average packet-in processing latency is minimized.

We represent the communication between the different application slices using a *communication graph* whose vertices are application slices. Thus, there are $P \times N$ vertices in this graph. The edges in the graph denote communication between slices. Communication can occur between two different applications in the same partition (e.g., packets permitted by a firewall module may then be forwarded to a load-balancer), as well as between two slices of the same application in different partitions (e.g., a bandwidth reservation application between a source and destination in two different partitions will require communication between the application slices in the two partitions).

Let d_{ij} denote the communication cost between two slices. Because we are interested in latency, the communication cost denotes the additional latency overhead if the slices are placed in different machines. Let A_i denote a vertex in the communication graph where $i \in [1, P \times N]$. Then, depending on placement, we have the vector $F[i] = k$ which denotes that application slice A_i is placed in machine k.

Objective function Next, we model latency of latency-sensitive events. Because these events typically traverse multiple application slices, event-handling latency would depend on the total communication cost across these applications slices (i.e., path delay). Let $E = \{e_1, e_2, ..., e_r\}$ be the events of interest, with their associated paths, $\{p_1, p_2, .., p_r\}$, in the communication graph. Naturally, each path is a sequence of edges in the graph.

Then, the cost of an event is given by:

$$\forall p_m \in P, \ t_{lat}(p_m) = \sum_{<i,j> \in p_m, F[i] \neq F[j]} d_{ij} \tag{3}$$

In this formulation, two slices would incur latency overhead of d_{ij} when placed in different servers but no overhead when co-located in the same physical machine.

We can assign a weight (e.g., relative priority, probability) to each event and calculate the weighted latency as follows.

$$t_{lat} = \sum_{p_m \in \{p_1, p_2, \dots p_r\}} \gamma(p_m) t_{lat}(p_m) \tag{4}$$

The weights could be relative priorities of the events based on semantic knowledge or could just be event probabilities. Our objective is to minimize equation (4) subject to capacity (i.e., CPU and memory), latency, and correctness constraints.

Capacity constraints Let the compute and memory capacity of each server be R_{cpu} and R_{mem}, respectively. Let A_i's compute and memory requirements be C_i and M_i, respectively. Then, we have the following constraints based on CPU and memory capacities.

$$\begin{aligned} \max_k \left(\sum_{\forall i: F[i]=k} C_i \right) \le R_{cpu} \\ \max_k \left(\sum_{\forall i: F[i]=k} M_i \right) \le R_{mem} \end{aligned} \tag{5}$$

real-time constraints We can bound the latency for real-time applications using an additional constraint of the form:

$$t_{lat}(p_m) <= deadline_m \tag{6}$$

where p_m is a path of a real-time event m in the graph.

3.4 Communication-aware placement: simplification

The final form of the objective function t_{lat} is the linear combination $t_{lat} = \sum_{F[i] \neq F[j]} \alpha_{ij} d_{ij}$, for some coefficients α_{ij}. If we ignore the constraints (i.e., equations (6) and (5), we see that t_{lat} only depends on the weight of the edge-cut between the partitions and our aim is to find such a mapping F. If we ignore only equation (6), the problem reduces to the well-known *multi-constraint graph partitioning* [15] problem. If each vertex A_i is assigned a vector of weights $\langle C_i, M_i \rangle$ denoting the compute and memory requirement of each slice, then the problem is *equivalent* to finding a *S-way* partitioning such that the partitioning satisfies a constraint associated with each weight, while attempting to minimize the weight of edge-cut. Because multi-constraint graph partitioning is a known NP-hard problem [15], we employ heuristic methods from [14] which deliver high quality results in reasonable time. While our heuristic solution ignores equation (6), we did not observe appreciable degradation in our experiments.

3.5 Discussion

We discuss dynamic load adaptation and fault tolerance.

Load adaptation Some previous papers ([7, 17]) argue for the controller's partitioning and placement to change according to instantaneous load from switches (e.g., packet-in rate). However, such dynamic re-partitioning and placement requires applications to re-partition and migrate their state which drastically affects controller performance and offsets the cost advantage of dynamic re-partitioning. This cost of reorganizing state applies to controllers that store state locally as well as to those that use a distributed datastore. While controllers that store state locally must aggregate/split/migrate their state whenever partitioning/placement changes [17], controllers that use a distributed datastore must reshard their datastore whenever the partitioning changes [7]. Because the cost of provisioning for the peak load is a small fraction (e.g., dedicating 100 servers for a 100,000-server datacenter is only 0.1%) of total cost of ownership (TCO) of large datacenters, we provision enough servers to accommodate the peak load and do not change our partitioning based on packet-in rate (load). Nevertheless, if desired, *Hydra's* placement algorithm is fast enough to respond to load variations.

Fault tolerance For fault tolerance reasons, it may be desirable to replicate SDN controllers in each partition, either using a simple master-slave design for each partition, or a more strongly consistent approach based on the Paxos algorithm [18]. While fault tolerance mechanisms are orthogonal to our work, it is easy to generalize *Hydra* to handle the placement of replicas. Specifically, a simple approach is to replicate the configuration produced in the previous section as many times as needed for adequate fault tolerance. If it is also desirable to consolidate the number of physical controller machines, our model could be extended by including additional variables for each replica, and using the same placement algorithms described in the previous section. To ensure that replicas of a given application/partition slice are not placed on the same physical host, additional constraints may be added to require replicas be placed in different hosts. Finally, there might be additional requirements that parts of the network supplied by different power sources need controller isolation for fault tolerance. This constraint can be added to our formulation by requiring that applications corresponding to these partitions not be co-located with each other.

4 Experimental Methodology

In this section, we present the details of our implementation and our evaluation methodology.

SDN Applications: We use the Floodlight SDN controller [2], which is a widely used OpenFlow controller. We evaluate four control-plane functions:

1. *Shortest path computation* (DJ): Shortest path computation based on Dijkstra's algorithm, which runs whenever a new link (switch) is discovered or an existing link (switch) fails.
2. *Firewall* (FW): Filters packet-in messages based on a set of rules.
3. *Route Lookup* (RL): Returns the complete path based on source/destination pair in a packet-in header.
4. *Heart-beat handler* (HB): Generates and forwards heart-beat messages between switches and controllers;

DJ is a *computationally intensive* intensive application; FW and RL are *latency-sensitive* applications and are invoked during path setup; HB is *real-time* application – if a heart-beat is not processed within a deadline (i.e., heart-beat interval), a spurious link/switch failure would result which would trigger DJ. While a production SDN deployment would include tens of applications, it is hard for researchers to study a large number of applications at production scales.

Load Generation: *Hydra's* evaluation requires large topologies with a few thousand switches. Because network emulators such as *Mininet* model both control and data plane, they do not scale beyond a few tens of switches [7]. Therefore, we use *CBench* [1]. CBench generates packet-in events that stress the control-plane without modeling a full-fledged data-plane. While the current implementation of *CBench* generates random packet-in messages (to potentially non-existent destinations), we modified CBench to generate packets that are meaningful to our topology. We use a reactive model of SDN in our experiments. However, our results are generalizable to both pro-active or reactive models.

Topology: Datacenters typically employ hierarchical topologies which provide high bisection bandwidth and good fault tolerance [3, 22, 13]. Our datacenter topology is a fat-tree with 2560 switches. The topology is organized into 512 core switches, and 32 pods, with each pod containing 32 Top of Rack (ToR) switches.

5 Results

In this section, we compare *Hydra* to *Topological slicing* for the three types of applications. Recall that we care about different metrics depending on the application type – lower missed heart-beats (deadlines) for real-time applications (HB), lower latency (higher throughput) for latency-sensitive applications (FW,RL), and lower convergence time for computationally-intensive applications (DJ).

We begin by showing how convergence time varies with the number of partitions which enables us to choose the right partition size. Then we show how our *communication-aware placement* co-locates different application slices. Because our placement depends on CPU and memory utilization, we show CPU and memory utilizations which are sensitive to a variety of parameters such as packet-in rates, topology sizes, and other parameters. After placement, we compare missed heart-beats for HB and throughput (at near-saturation high loads, throughput is a *proxy* for latency as queuing becomes the dominant latency component) for FW and RL.

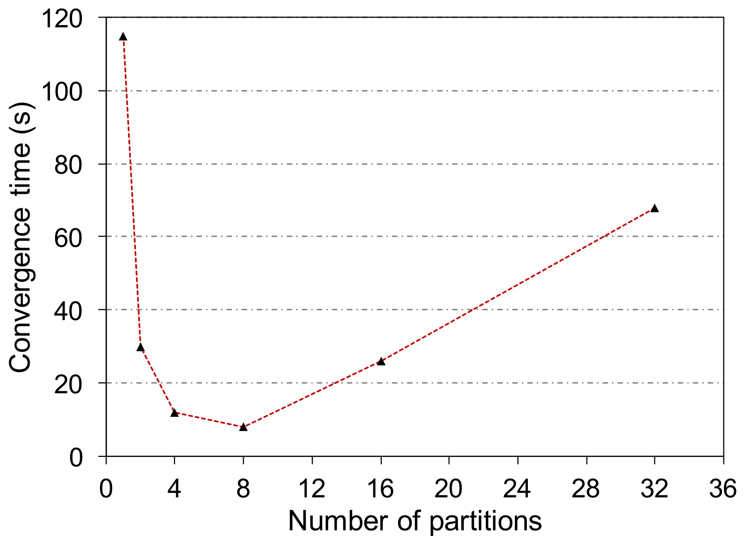


Fig. 4: Convergence time

5.1 Convergence Time

We study convergence time for our fat-tree topology with 2560 switches. Because fat-tree is hierarchical, it is straight-forward to create partitions by grouping neighboring pods. For example, we can create two partitions by grouping 16 pods in one partition and the other 16 in the other partition (each pod contains 32 ToR switches). Recall that convergence time is the time to recalculate shortest paths after a link failure. So, to measure convergence time, we take down a *random* link in our fat-tree which could be a border link (i.e., core link) or a partition-local link (i.e., ToR or aggregate links). We then measure the time required for *all* partitions to recompute their paths which includes time for inter-partition communication. While all neighboring partitions need to recompute on a border-link failure, a local link failure might also require partitions to advertise new costs to other partitions similar to BGP. For each partition size, we simulate 100 random link failures.

We show the average convergence time for DJ vs. number of partitions (partition size) in Figure 4. We vary the number of partitions (ToR switches per partition) as 1 (1024), 2(512), 4(256), 8(128), 16(64), and 32(32) along X-axis and show convergence time along Y-axis. We see that convergence time decreases *rapidly* as we increase the number of partitions from 1 to 8 due to amortization of compute from parallelization. However, after 8, convergence time starts to climb as communication overhead overwhelms gains from parallelization. Because topological slicing co-locates other applications with DJ, higher number of partitions are needed to accommodate the aggregate CPU and memory requirements. In contrast, Hydra's *functional slicing* enables us to choose the best partition size (e.g., 8 in this case), *independent* of other applications.

5.2 Communication-aware placement

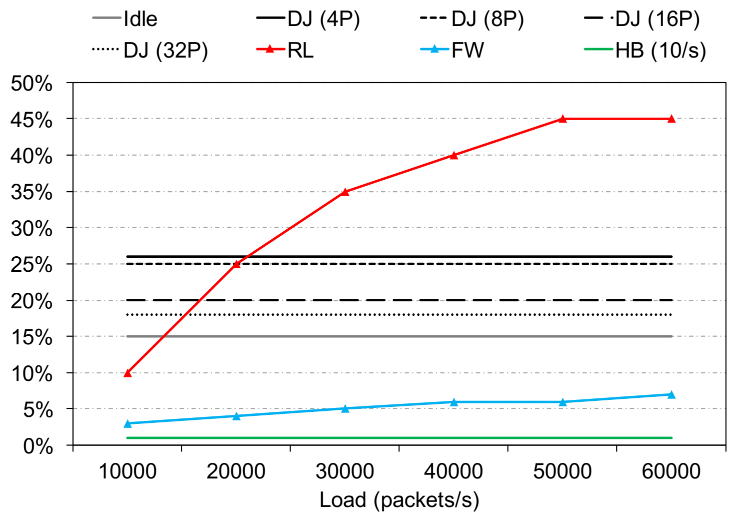


Fig. 5: Average CPU requirements

We start by showing the CPU and memory demands of applications. For these measurements, we ran Floodlight controller on our machine with 4 cores of CPU and 64 GB of memory. The demand of each application depends on the amount of application state and controller's load. Application state impacts both CPU and memory usage – applications maintain state in memory and look up state for each packet-in message. RL must keep local topology information which depends on the partition size. The number of firewall rules impacts FW's state overhead. In our experiments, we use $50,000$ firewall rules which is typical for large networks. DJ maintains both local and global topology information. DJ's CPU usage depends on link failure rate and partition size. We simulate a random link failure every 10 seconds which is reasonable for large networks. From Figure 4, we expect that DJ's CPU usage to be *highly* sensitive to partition size. HB's CPU and memory usage are minimal – its CPU usage slowly grows with heart-beat frequency but negligible overall.

The CPU demands of applications also depend on load (i.e., rate at which the controller receives packet-in messages from switches). We modified *CBench* to precisely control packet rate. Our base controller saturates around 50,000 packets per second. Therefore, we make measurements from 10,000 to 50,000 packets per second. Even without any applications, SDN controllers run some common functions (e.g., south-bound *OpenFlow* protocol handlers) which cannot be turned off. Therefore, we initially measure the idle CPU and memory usage without any applications (no incoming packets to the controller) which represents the overhead of starting a new controller instance. The overhead is about 15% CPU usage and $512MB$ of memory. We enable applications one-by-one and measure CPU and memory usage for each application (excluding idle overhead) at 100 ms intervals for 30 *seconds*. We discard initial and final samples to capture *steady-state* usage.

Figure 5 shows the CPU requirements of different applications as as we vary the load. DJ and HB do not depend on load – DJ's CPU usage depends on partition size and link failure rate (1 every 10 seconds), and HB's usage depends on heart-beat frequency (we ran HB at 10/second and 100/second but they are both insignificant). We show DJ for varying partition sizes – for example, DJ(4P) is for 4 partitions each with one fourth the number of switches as DJ(1P). We observe that DJ's CPU usage reduces with increasing number of partitions due to reduced number of switches. As discussed in the previous section, with topological slicing, the state overheads of other applications (e.g., RL, FW) determine the partition size which negatively impacts convergence time. For instance, we can see that the combined CPU usage of *Idle, RL, FW, HB, and DJ(4P)* is close to 100% (15+45+7+3+25) for higher loads. In fact, only when there are more than 8 partitions, the combined CPU usage falls well below 100% (servers usually operate at less than 90% loads to provide reasonable response times). Therefore, topological slicing is forced to choose a partition size of 16 or more which leads to high convergence times (see Figure 4). *Hydra*, on the other hand, separates DJ from other applications, enabling DJ to use the *best* partition size.

Memory usage is largely independent of load. Table 1 shows the average memory overheads of DJ, FW, and RL for the one partition case containing all switches. From the table, it is clear that memory does not impact our placement in our controller as all of applications comfortably fit within our memory capacity. However, we expect production controllers to have large state overheads that will not fit within one server's memory. We do not show HB's memory overhead as it is negligible.

Table 1: Memory requirements

DJ	**RL**	**FW**
6.25 GB	3.75 GB	1.25 GB

Recall from section 4 that our communication graph has only one edge between RL and FW, as RL and FW are the only applications that lie in the critical path of flow's path setup; DJ and HB do not have edges between them or to either RL or FW. From Figure 5 and table 1, it is straight forward to see the difference between Topological slicing's and Hydra's placement decisions. Topological partitioning requires 16 controller instances (16 partitions) requiring 16 cores. Each instance would host all the applications. In contrast, *Hydra* creates 8 network partitions (*minima* in Figure 4). For each partition, it assigns two controller instances which run on separate CPU cores. While one controller instance hosts DJ for that partition, another instance hosts all the *other* applications – RL, FW, and HB. While we could manually calculate optimal placements in this simple controller, deployment-scale controllers would likely consist of tens of applications with complex communication patterns, and, therefore, would require a rigorous approach such as Hydra. Unfortunately, it is harder for researchers to experiment with production-scale controllers without access to production-scale networks and workloads.

5.3 Latency-sensitive applications

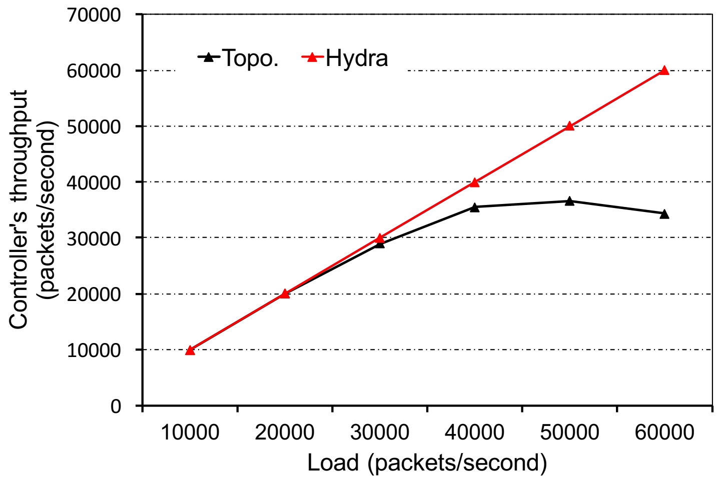


Fig. 6: Scalability of latency-sensitive applications in Hydra

In this experiment, we compare the performance of latency-sensitive applications in *one network partition*. Recall that Hydra creates 8 network partition ($1/8^{th}$ switches) as opposed to topological slicing which creates 16 partitions ($1/16^{th}$ switches). In Figure 6, we compare the scalability of latency-sensitive applications in Hydra vs. topological slicing. We show load (injected packets per second) along X-axis and the achieved throughput after route lookup (RL) and firewall processing (FW) along Y-axis. As we can see, Hydra scales well beyond $60,000$ packets per second whereas topological slicing saturates at about $40,000$. As a result, latency-sensitive events incur high queuing inside the controller in the case of topological slicing. It is also interesting to note that even though Hydra handles events from a larger number of switches, the latency-sensitive applications (RL and FW) are isolated from the load spikes caused by computationally-intensive DJ application, thanks to *functional slicing*.

5.4 Real-time applications

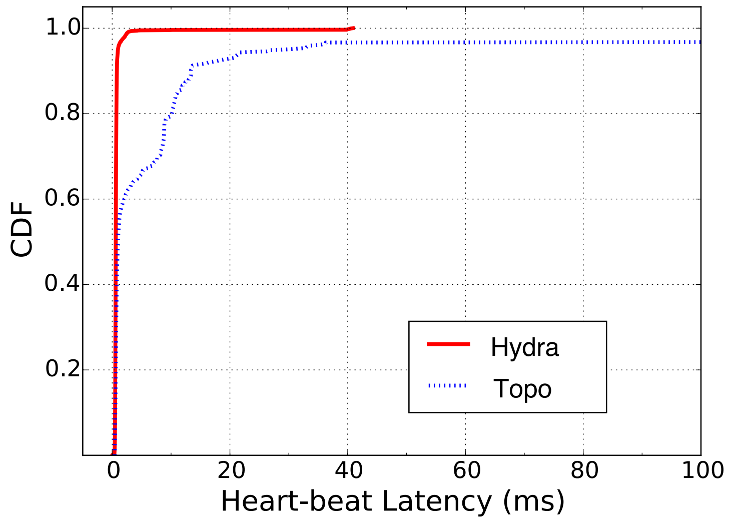


Fig. 7: Performance of real-time apps. in Hydra

Separating computationally-intensive DJ application also helps our real-time heart-beats (HB) application. Figure 7 shows the CDF of heart-beat latency between Hydra and topological slicing. Our default heart-beat frequency is 10 heart-beats per second. We see a marked difference between the two – while *Hydra's* 95^{th} and 99^{th} %-iles are about 10 ms, topological slicing's 95^{th} %-ile is about 30 ms. With a deadline of 100 ms (i.e., periodicity of heart-beats), topological slicing would suffer about 3% missed deadlines, whereas Hydra would not miss *any*. While 3% may look like a small number, but penalty for missed deadlines is very high (i.e., missed deadlines trigger expensive path recomputation which would further exacerbate the problem).

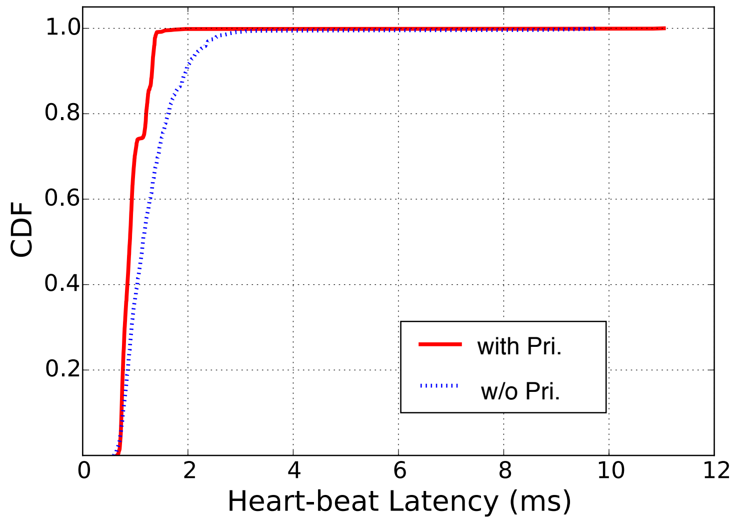


Fig. 8: Isolation of prioritization's gains

5.5 Isolating the impact of prioritizing

In this section, we isolate the gains from prioritizing real-time applications over latency-sensitive applications. In Figure 8, we compare the CDF of heart-beat latency between Hydra with and without prioritization. In this experiment, we use a heart-beat frequency of 10 heart-beat messages per second. Further, we pump packet-in messages such that RL operates close to its saturation in terms of packets per second that can be processed by RL. We show the heart-beat processing latency in *milliseconds* along X-axis and cummulative percent of requests along Y-axis. We observe that the responses are received in a timely fashion when HB is prioritized over RL (i.e., Hydra). However, when heart-beats are not prioritized (i.e., topological slicing), the performance modestly degrades. As we increase the heart-beat frequency, which is desired for quicker failure detection, prioritizing becomes even more critical as we show next.

5.6 Sensitivity to heart-beat frequency

In this experiment, we increase the heart-beat frequency to 100 per second to facilitate quicker failure detection (While 10 heart-beats per second implies a failure detection time of 100 *ms*, 100 heart-beats per second implies a failure detection time of 10 *ms*). Similar to the previous experiment, we pump packet-in messages such that RL operates close to its saturation in terms of packets per second that can be processed by RL. Similar to figure 8, figure 9 shows heart-beat processing latency in *milliseconds* along X-axis and cummulative percent of requests along Y-axis. We see that *almost* all HB messages meet the deadline

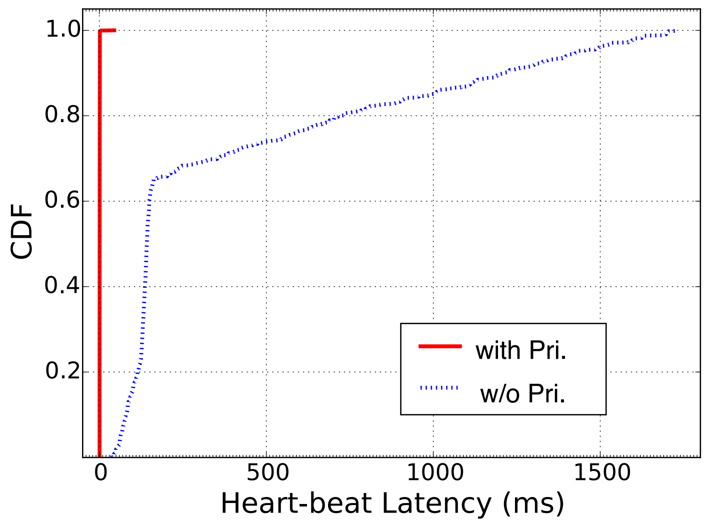


Fig. 9: Sensitivity to heart-beat rate

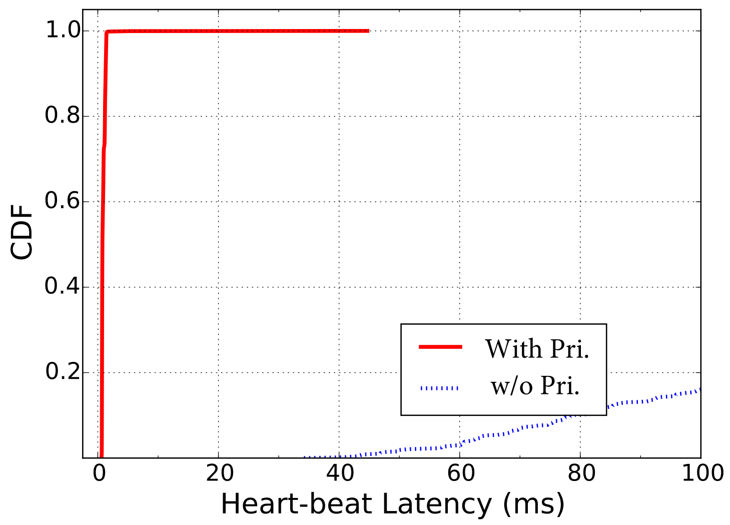


Fig. 10: Sensitivity to heart-beat rate (X-axis truncated to 100ms)

when prioritized but *no* messages meet the deadline when not prioritized. In fact, some HB messages take as long as 1800 *ms* to get a response. Figure 10 which shows the same data as Figure 9 but truncates the X-axis to 100 ms to show highlight the impact of prioritization. These results clearly show that it is better not to co-locate HB and RL (i.e., functional slicing helps) unless a less aggressive failure detection is acceptable.

6 Related Work

SDN has received a lot of attention from the research community over the last several years. While there is a plethora of work that cover many aspects of SDN, a systematic analysis of controller partitioning and placement is not well-studied. Onix [16] focuses on providing APIs for control-plane and state distribution. Beehive [23] enables applications to express their state-dependence and uses the inferred state-dependence to co-locate functions *within* each application. In contrast, Hydra considers event-processing pipeline across applications and considers others constraints (e.g., CPU load, memory) to partition applications as well as the state (i.e., topology).

Hyperflow [21] improves controller performance by pro-actively synchronizing state but does not deal with partitioning. Kandoo [9] offloads switch-local events to switches but does not address a large subset of events that are not local to the switch. ElastiCon [7] topologically partitions the controller based on CPU load. In contrast, Hydra employs a hybrid of topological and functional partitioning. A few other papers address the placement of the controller on the network to reduce network delays and to topologically-slice the network for better performance [20, 10]. But none of them employ functional slicing and they do not target specific response times and convergence costs. While some papers [16, 7] argue for a logically-separate, globally-consistent, distributed datastore for storing state to ease communication among different controllers, others [17] prefer that the state be distributed among controller instances like many distributed or parallel applications today. Nevertheless, our optimization formulation is agnostic to the choice of state management. In our evaluation, we use Floodlight [2] which assumes the latter alternative where there is no separate datastore but other communication costs (e.g., datastore) can be easily incorporated into our model.

7 Conclusion

In this paper, we have presented *Hydra*, a framework for distributing SDN control functions across servers. *Hydra* combines well-known topological slicing with our novel *functional slicing* and distributes applications based on their communication pattern. We have demonstrated the importance of functional slicing and *communication-aware* placement in the scalability of SDN with extensive evaluations.

Our results, while promising, are only a start. First, while we evaluated using applications that are available publicly controllers, we expect *Hydra's* benefits to be even higher with large-scale deployments. Getting access to production SDN deployments can enable larger-scale evaluations, which is an interesting direction for future work. Second, we are building a more comprehensive system based on functional slicing, that can handle other issues such as incrementally modifying application placement as loads drastically change and incorporating consistency guarantees into the model.

References

1. Controller benchmark. `http://www.openflowhub.org/display/floodlightcontroller/Cbench`
2. Floodlight. `http://www.projectfloodlight.org`
3. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. In: Proceedings of the ACM SIGCOMM 2008. pp. 63–74 (2008)
4. Chang, Y., Rezaei, A., Vamanan, B., Hasan, J., Rao, S., Vijaykumar, T.: Hydra: Leveraging functional slicing for efficient distributed sdn controllers. In: Proceedings of the International Conference on Communication Systems and Networks (COMSNETS). pp. 1–8 (Jan 2017)
5. Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S.: Devoflow: Scaling flow management for high-performance networks. In: Proceedings of the ACM SIGCOMM. pp. 254–265 (2011)
6. Demetrescu, C., Eppstein, D., Galil, Z., Italiano, G.F.: Algorithms and theory of computation handbook. chap. Dynamic Graph Algorithms, pp. 9–9 (2010)
7. Dixit, A.A., Hao, F., Mukherjee, S., Lakshman, T., Kompella, R.: Elasticon: An elastic distributed sdn controller. In: Proceedings of the ANCS. pp. 17–28 (2014)
8. Greenberg, A., Hjalmtysson, G., Maltz, D.A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., Zhang, H.: A clean slate 4d approach to network control and management. SIGCOMM Comput. Commun. Rev. 35(5), 41–54 (2005)
9. Hassas Yeganeh, S., Ganjali, Y.: Kandoo: A framework for efficient and scalable offloading of control applications. In: Proceedings of the HotSDN. pp. 19–24 (2012)
10. Heller, B., Sherwood, R., McKeown, N.: The controller placement problem. In: Proceedings of HotSDN. pp. 7–12 (2012)
11. Hong, C.Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., Wattenhofer, R.: Achieving high utilization with software-driven wan. In: Proceedings of the ACM SIGCOMM. pp. 15–26 (2013)
12. Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., Vahdat, A.: B4: Experience with a globally-deployed software defined wan. In: Proceedings of the ACM SIGCOMM. pp. 3–14. ACM (2013)
13. Kabbani, A., Vamanan, B., Hasan, J., Duchene, F.: Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In: Proceedings of CoNEXT. pp. 149–160 (2014)
14. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20(1), 359–392 (Dec 1998)
15. Karypis, G., Kumar, V.: Multilevel algorithms for multi-constraint graph partitioning. In: Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA. p. 28 (1998)

16. Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., Shenker, S.: Onix: A distributed control platform for large-scale production networks. In: Proceedings of OSDI. pp. 1–6 (2010)
17. Krishnamurthy, A., Chandrabose, S.P., Gember-Jacobson, A.: Pratyaastha: An efficient elastic distributed sdn control plane. In: Proceedings of the HotSDN. pp. 133–138. ACM, New York, NY, USA (2014)
18. Lamport, L.: Paxos made simple. ACM Sigact News 32(4), 18–25 (2001)
19. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38(2), 69–74 (2008)
20. Tam, A.W., Xi, K., Chao, H.: Use of devolved controllers in data center networks. In: INFOCOM WKSHPS. pp. 596–601 (April 2011)
21. Tootoonchian, A., Ganjali, Y.: Hyperflow: A distributed control plane for openflow. In: Proceedings of INM/WREN. pp. 3–3 (2010)
22. Vamanan, B., Hasan, J., Vijaykumar, T.: Deadline-aware datacenter tcp (d2tcp). In: Proceedings of the ACM SIGCOMM 2012. pp. 115–126 (2012)
23. Yeganeh, S.H., Ganjali, Y.: Beehive: Towards a simple abstraction for scalable software-defined networking. In: Proceedings of HotNets-XIII. pp. 13:1–13:7 (2014)